

Type-Based Enforcement of Secure Programming Guidelines

Code Injection Prevention at SAP

Robert Grabowski¹ Martin Hofmann¹ Keqin Li²

¹Ludwig-Maximilians-Universität München

²SAP Research Sophia-Antipolis

FAST Workshop · Leuven, Belgium · September 15-16, 2011

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");  
output("<script>");  
output("  alert('" + name + "');");  
output("</script>");
```

- name may close ' ' quotes of alert, include malicious code

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");  
output("<script>");  
output("  alert(''); form.submit('http://...');");  
output("</script>");
```

- name may close ' ' quotes of alert, include malicious code

Problem: Code Injection

caused by erroneous string handling

- allows introduction of malicious code into strings that are interpreted/executed
- example: generate HTML page with data from GET request:

```
String name = request.getInputParameter("name");  
output("<script>");  
output("  alert(''); form.submit('http://...');");  
output("</script>");
```

- name may close ' ' quotes of alert, include malicious code

very common vulnerability, comes in many forms

Countermeasures Available to the Programmer

static program analysis

- dataflow analysis, information flow analysis, . . .
- may be hard to understand and use; tools too specialized

runtime protection

- “tainted” tags, no cross-domain execution, . . .
- very interpreter-dependent; runtime overhead

most common in practice: programming guideline

- easy to understand “best practice” for safe programming

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");  
output("<script> alert('" + name + "'); </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or “best practice” for the programmer:
always use escape function on untrusted strings that are embedded in the output

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");  
output("<script> alert('" +escapeToJs(name)+ "') </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or “best practice” for the programmer:
always use escape function on untrusted strings that are embedded in the output

Guideline for Usage of SAP Sanitization Framework

```
String name = request.getInputParameter("name");  
output("<script> alert('" +escapeToJs(name)+ "') </script>");
```

- provide method `escapeToJs(name)` that eliminates ' ' quotes in string `name`
- guideline or “best practice” for the programmer:
always use escape function on untrusted strings that are embedded in the output

obvious problem: programmer is responsible for following guideline

Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a Java programmer follows a given programming guideline .

Contribution: Type-Based Enforcement of the Guideline

We provide a **type system** that ensures that a Java programmer follows a given programming guideline .

- types: familiar to programmers; extension of class types

Contribution: Type-Based Enforcement of the Guideline

We provide a type system that ensures that a **Java programmer** follows a given programming guideline .

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code

Contribution: Type-Based Enforcement of the Guideline

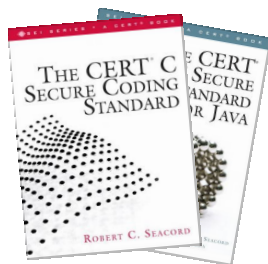
We provide a type system that ensures that a Java programmer follows a given programming guideline .

- types: familiar to programmers; extension of class types
- working tool: analysis of real Java code
- guideline: correct usage of SAP sanitization framework

Vision: Programming Guideline Adherence Checking

programming guidelines

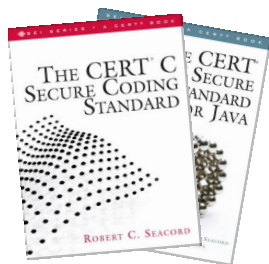
- condensed expert knowledge, easy to apply for programmer
 - cover many topics: memory safety, scheduling safety, resource handling, ...
 - often described informally
- ubiquitous in practice, but rarely research focus



Vision: Programming Guideline Adherence Checking

programming guidelines

- condensed expert knowledge, easy to apply for programmer
 - cover many topics: memory safety, scheduling safety, resource handling, ...
 - often described informally
- ubiquitous in practice, but rarely research focus



automatic support for implementing guidelines

- pick and formalize existing programming guideline
- provide easy-to-use tools that check adherence to guideline
- separate goal:
evaluate guideline with respect to security properties

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
    String input = request.getInputParameter();

    // case 1: HTML embedding
    String s = "<body>" + escapeToHtml(input) + "</body>";
    output(s);

    // case 2: JavaScript embedding
    output("<script>");
    output("    alert('" + escapeToJs(input) + "');");
    output("</script>");
}
```

ensure use of escape methods, depending on embedding position

Concrete application scenario: Java servlet

```
public void doGet(HttpServletRequest request) {
    String input = request.getInputParameter();

    // case 1: HTML embedding
    String s = "<body>" + escapeToHtml(input) + "</body>";
    output(s);

    // case 2: JavaScript embedding
    output("<script>");
    output("    alert('" + escapeToJs(input) + "');");
    output("</script>");
}
```

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
    String input = request.getInputParameter();

    // case 1: HTML embedding
    String s = "<body>" + escapeToHtml(input) + "</body>";
    output(s);

    // case 2: JavaScript embedding
    output("<script>");
    output("    alert('" + escapeToJs(input) + "');");
    output("</script>");
}
```

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {  
    String input = request.getParameter("input");  
  
    // case 1: HTML embedding  
    String s = "<body>" + escapeToHtml(input) + "</body>";  
    output(s);  
  
    // case 2: JavaScript embedding  
    output("<script>");  
    output("    alert('" + escapeToJs(input) + "');");  
    output("</script>");  
}
```

C1: sanitized for HTML

Literal: trusted

Literal

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {
    String input = request.getParameter("input");

    // case 1: HTML embedding
    String s = "<body>" + escapeToHtml(input) + "</body>";
    output(s);

    // case 2: JavaScript embedding
    output("<script>");
    output("    alert('" + escapeToJs(input) + "');");
    output("</script>");
}
```

C1: sanitized for HTML

Literal: trusted

Literal

<script>: enters JS mode

</script>: leaves JS mode

C2: sanitized for JavaScript

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {  
    String input = request.getParameter();  
  
    // case 1: HTML embedding  
    String s = "<body>" + escapeToHtml(input) + "</body>";  
    output(s);  
}
```

C1: sanitized for HTML

Literal: trusted

Literal

```
    // case 2: JavaScript embedding  
    output("<script>");  
    alert("'" + escapeToJs(input) + "'");  
    output("</script>");  
}
```

<script>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

</script>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {  
    String input = request.getParameter("input");  
  
    // case 1: HTML embedding  
    String s = "<body>" + escapeToHtml(input) + "</body>";  
    output(s);  
}
```

C1: sanitized for HTML

Literal: trusted

Literal

```
    // case 2: JavaScript embedding  
    output("<script>");  
    alert("'" + escapeToJs(input) + "'");  
    output("</script>");  
}
```

<script>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

</script>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents
- calls to output induce abstract output sequence

Concrete application scenario: Java servlet

Input: untrusted

```
public void doGet(HttpServletRequest request) {  
    String input = request.getParameter("input");  
  
    // case 1: HTML embedding  
    String s = "<body>" + escapeToHtml(input) + "</body>";  
    output(s);  
}
```

C1: sanitized for HTML

Literal: trusted

Literal

```
    // case 2: JavaScript embedding  
    output("<script>");  
    alert("'" + escapeToJs(input) + "'");  
    output("</script>");
```

<script>: enters JS mode

Literal

Literal

C2: sanitized for JavaScript

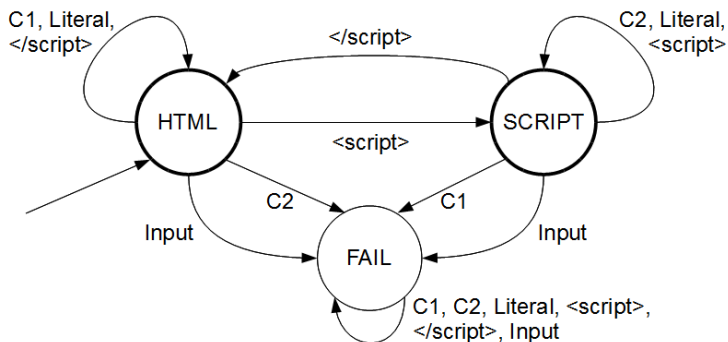
</script>: leaves JS mode

ensure use of escape methods, depending on embedding position

- classify strings according to origin and contents
- calls to output induce abstract output sequence
- define set of allowed output sequences as regular language

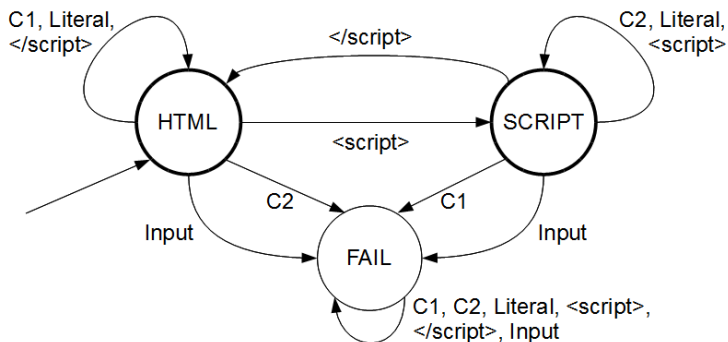
Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet $\Sigma = \text{classifications}$



Guideline expressed as finite state machine

automaton $G = (Q, q_0, \delta, F)$ with alphabet $\Sigma = \text{classifications}$



- accepted words = allowed program output
- type system is defined with respect to given automaton

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	SCRIPT	FAIL
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	SCRIPT	FAIL
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

- $[w]$ is allowed if $\delta(q_0, w) \in F$

Compact representation of accepted traces

Construction of syntactic monoid

- $w_1 \equiv w_2$ if for all $q \in Q$, $\delta(q, w_1) = \delta(q, w_2)$
- defines equivalence classes:

	HTML	SCRIPT	FAIL
[Literal]	HTML	SCRIPT	FAIL
[C1]	HTML	FAIL	FAIL
[C2]	FAIL	SCRIPT	FAIL
[<script>]	SCRIPT	SCRIPT	FAIL
[</script>]	HTML	HTML	FAIL
[Input]	FAIL	FAIL	FAIL
[C1<script>]	SCRIPT	FAIL	FAIL
[C2</script>]	FAIL	HTML	FAIL

- $[w]$ is allowed if $\delta(q_0, w) \in F$
- class concatenation: $[w_1][w_2] = [w_1 w_2]$

String analysis for FJEUS

- FJEUS:

Featherweight Java with Extended Updates and Strings

```
e ::= x | let x = e1 in e2 | if x1 = x2 then e1 else e2 |  
    null | new C | x.f | x1.f := x2 | x.m( $\bar{x}$ ) |  
    "str" | x1 + x2
```

String analysis for FJEUS

- FJEUS:
Featherweight Java with Extended Updates and Strings

$$e ::= x \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{if } x_1 = x_2 \text{ then } e_1 \text{ else } e_2 \mid \\ \text{null} \mid \text{new } C \mid x.f \mid x_1.f := x_2 \mid x.m(\bar{x}) \mid \\ \text{"str"} \mid x_1 + x_2$$

- operational semantics with output trace / effect

$$(s, h) \vdash e \Downarrow v, h' \ \& \ w_t$$

- values v : object location
- traces w_t : word over classifications alphabet Σ

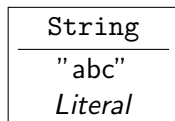
Instrumented string semantics

expression

value

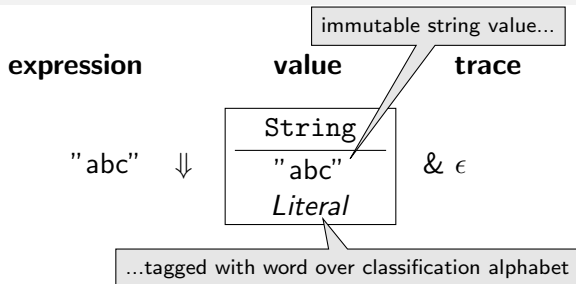
trace

"abc"

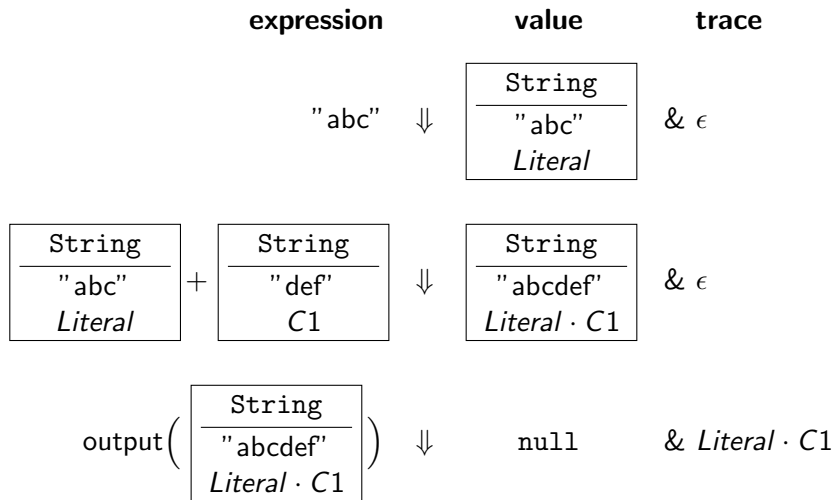


& ϵ

Instrumented string semantics



Instrumented string semantics



Type and effect system

FJ type system refined with annotated string types:

- defines typing judgement: $\Gamma \vdash e : \tau \& U_t$
- type $\tau \in \text{String}_U, \text{Object}, \text{Integer}, \dots$
- annotations U, U_t are sets of classes $[w]$

Type and effect system

FJ type system refined with annotated string types:

- defines typing judgement: $\boxed{\Gamma \vdash e : \tau \& U_t}$
- type $\tau \in \text{String}_U, \text{Object}, \text{Integer}, \dots$
- annotations U, U_t are sets of classes $[w]$

Interpretation

If $(s, h) \vdash e \Downarrow v, h' \& w_t$ and $\Gamma \vdash e : \text{String}_U \& U_t$

...and a couple of other premises...

then

- v points to string object tagged with w such that $[w] \in U$
- $[w_t] \in U_t$

Type and effect system

- typing rules approximate instrumented semantics:

String
"abc"
C1

 : String_{[C1],[C2]}

Type and effect system

- typing rules approximate instrumented semantics:

$$\boxed{\begin{array}{c} \text{String} \\ \hline \text{"abc"} \\ C1 \end{array}} : \text{String}_{\{[C1],[C2]\}}$$

- external functions are assigned a trusted signature:

getInputParameter() : $\text{String}_{\{[Input]\}} \& \{[\epsilon]\}$

escapeToHtml($x : \text{String}_{\{[Input]\}}$) : $\text{String}_{\{[C1]\}} \& \{[\epsilon]\}$

output($x : \text{String}_U$) : $\text{Void} \& U$ for all $U \subseteq M$

Example: rejected program

```
main () : Void =
```

```
  let u = getInputParameter() in
```

```
  let e = escapeToJs(u) in
```

```
  output(e)
```

Example: rejected program

main () : *Void* =

let $u = \text{getInputParameter}()$ in

effect = $\{\{\epsilon\}\}$

$\Gamma(u) = \text{String}\{\{Input\}\}$

let $e = \text{escapeToJs}(u)$ in

output(e)

Example: rejected program

main () : *Void* =

let *u* = *getInputParameter*() in

effect = {[ϵ]}

$\Gamma(u) = \text{String}_{\{\text{Input}\}}$

let *e* = *escapeToJs*(*u*) in

effect = {[ϵ]}

$\Gamma(e) = \text{String}_{\{C2\}}$

output(*e*)

Example: rejected program

main () : *Void* =

let *u* = *getInputParameter*() in

effect = {[ϵ]}

$\Gamma(u) = \text{String}\{\{Input\}\}$

let *e* = *escapeToJs*(*u*) in

effect = {[ϵ]}

$\Gamma(e) = \text{String}\{\{C2\}\}$

output(*e*)

effect = {[*C2*]}

Example: rejected program

main () : *Void* =

let *u* = *getInputParameter*() in

effect = {[ϵ]}

$\Gamma(u) = \text{String}_{\{\text{Input}\}}$

let *e* = *escapeToJs*(*u*) in

effect = {[ϵ]}

$\Gamma(e) = \text{String}_{\{C2\}}$

output(*e*)

effect = {[*C2*]}

overall effect: [ϵ][ϵ][*C2*] = [*C2*]

→ effect is not allowed (leads to FAIL state)

Inference of string type annotations and effects

inference algorithm based on declarative type system

- for each string variable and effect, create type variable U
- typing rules generate set constraints; solved externally

improved precision by context-sensitivity: [Beringer, G., Hofmann '10]

- polymorphic method types used for different call contexts
- type system is parametric in chosen context abstraction

Implementation

based on Java compiler implemented in Ocaml [Tse & Zdancewic '05]

specify refined string types and method effects for framework API

```
interface API {  
    @ST("Input") String getInputParameter();  
    @ST("C1") String escapeToHtml(@ST("Input") String str);  
    @SE("C1") void output(@ST("C1") String str);  
}
```

no annotations required for actual code; all types and effects are inferred

```
String s = api.escapeToHtml(input);
```

inference: set constraints as Datalog rules; solved with Succinct Solver

[Nielson, Seidl, Nielson '03]

online demo at <http://jsa.tcs.ifi.lmu.de/>

Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

programming guideline

prevents ↓

cross-site scripting

Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

output trace accepted by
finite state machine

formalizes ↓

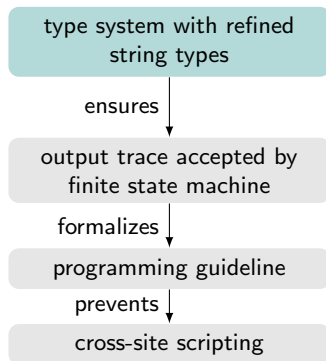
programming guideline

prevents ↓

cross-site scripting

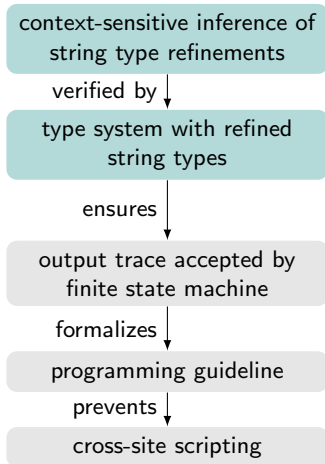
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks



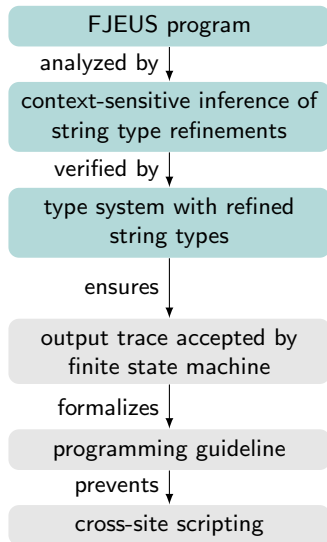
Summary

type-based enforcement of
string handling guidelines
to prevent XSS attacks

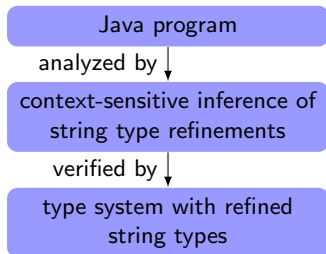


Summary

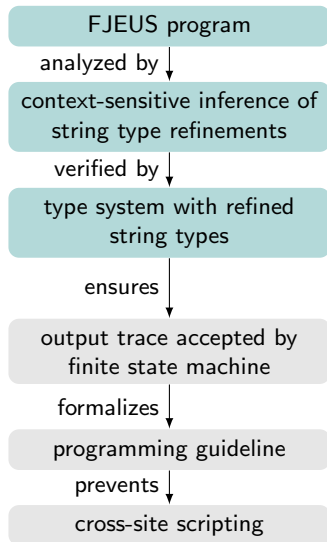
type-based enforcement of
string handling guidelines
to prevent XSS attacks



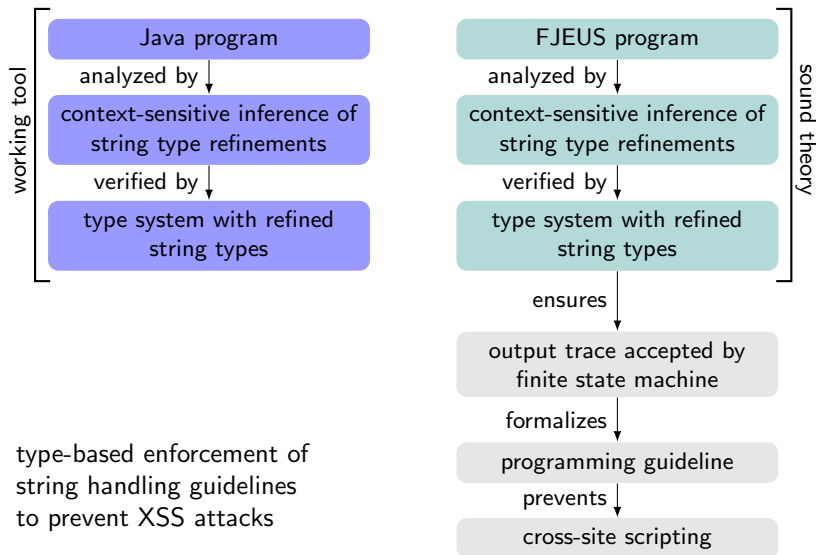
Summary



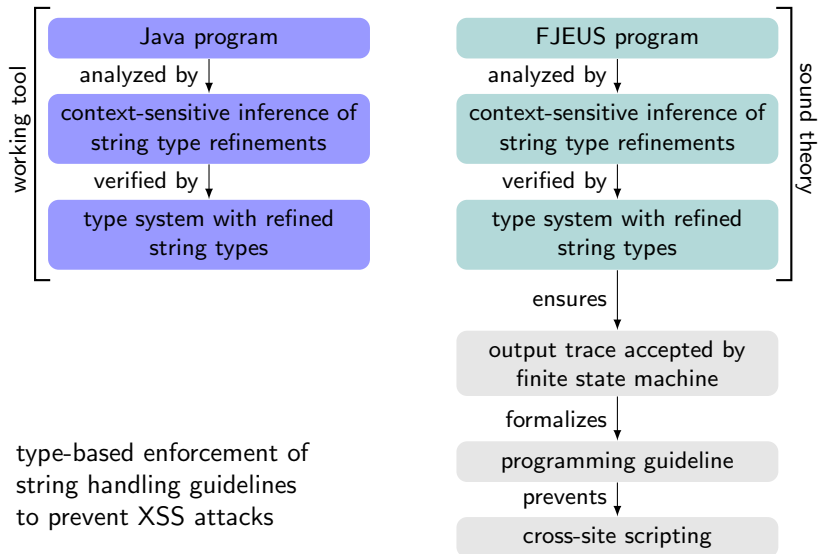
type-based enforcement of string handling guidelines to prevent XSS attacks



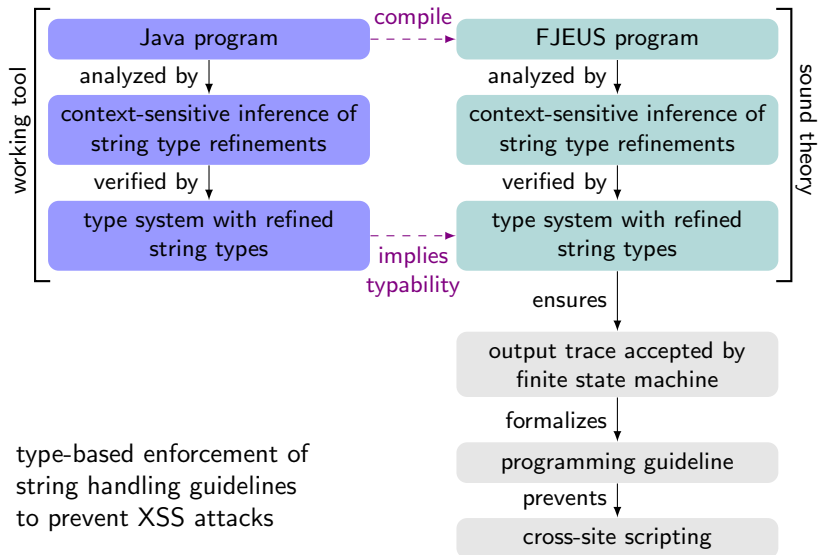
Summary



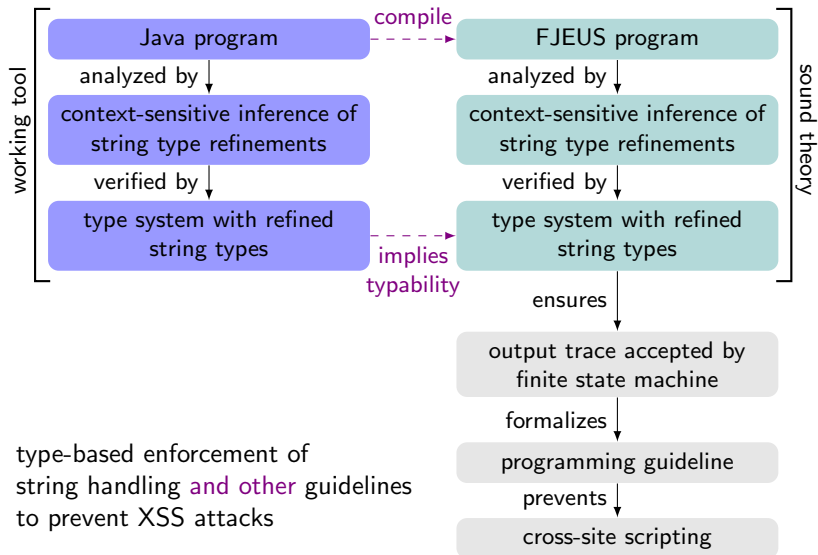
Summary and Future Work



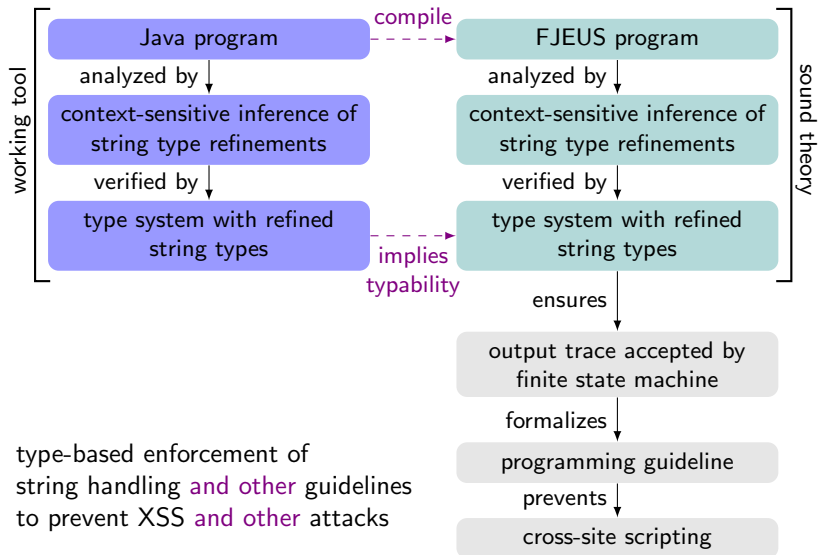
Summary and Future Work



Summary and Future Work



Summary and Future Work



Thank you for your attention!

<http://jsa.tcs.ifi.lmu.de/>

Backup slides: Actual use cases

- 1 embed between HTML tags:

```
output("<h1> Hello " + escapeToHtml(x) + "</h1>");
```

- 2 embed as attribute value:

```
output("<img alt='" + escapeToAttr(x) + "' />");
```

- 3 embed as URL attribute value:

```
output("<img src='http://' + escapeToUrl(x) + "' />");
```

- 4 embed within JavaScript:

```
output("<script>");  
output("  alert('" + escapeToJs(x) + "');");  
output("</script>");
```

Instrumented string semantics (extract)

$$\frac{(s, h) \vdash e_1 \Downarrow v_1, h_1 \ \& \ w_1 \quad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \ \& \ w_2}{(s, h) \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2, h_2 \ \& \ w_1 \cdot w_2}$$

$$\frac{l \notin \text{dom}(h) \quad h' = h[l \mapsto (\text{Script}, \langle \text{script} \rangle)]}{(s, h) \vdash \langle \text{script} \rangle \Downarrow l, h' \ \& \ \epsilon}$$

$$\frac{h(s(x_1)) = (w_1, \text{str}_1) \quad h(s(x_2)) = (w_2, \text{str}_2) \quad l \notin \text{dom}(h) \quad h' = h[l \mapsto (w_1 \cdot w_2, \text{str}_1 \cdot \text{str}_2)]}{(s, h) \vdash x_1 + x_2 \Downarrow l, h' \ \& \ \epsilon}$$

$$\frac{h(s(x)) = (w, -)}{(s, h) \vdash \text{output}(x) \Downarrow \text{null}, h \ \& \ w}$$

Typing rules (extract)

$$\overline{\Gamma \vdash \text{"abc"} : \text{String}_{\{\{\epsilon\}\}} \& \{\{\epsilon\}\}}$$

$$\overline{\Gamma \vdash \text{"<script>"} : \text{String}_{\{\{Script\}\}} \& \{\{\epsilon\}\}}$$

$$\frac{\Gamma \vdash e : \text{String}_U \& V \quad U \subseteq U' \quad V \subseteq V'}{\Gamma \vdash e : \text{String}_{U'} \& V'}$$

$$\frac{\Gamma(x_1) = \text{String}_U \quad \Gamma(x_2) = \text{String}_{U'}}{\Gamma \vdash x_1 + x_2 : \text{String}_{UU'} \& \{\{\epsilon\}\}}$$

$$\frac{\Gamma(x) = \text{String}_U}{\Gamma \vdash \text{output}(x) : \text{Void} \& U}$$

$$\frac{\Gamma \vdash e_1 : \tau \& V \quad \Gamma, x : \tau \vdash e_2 : \tau' \& V'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau' \& VV'}$$

Improving precision of the analysis

building on earlier work [Beringer, G., Hofmann '10]

- ① context-sensitive method analysis
- ② class types refined with regions

Context-sensitive method analysis

```
String appendLn(String s) { return s + "\n"; }
```

```
16 String x = appendLn("Your name is:");  
17 String y = appendLn(sanitizedInput);
```

$$\text{appendLn} : \text{String}_{\{\epsilon, [C1]\}} \rightarrow \text{String}_{\{\epsilon, [C1]\}}$$

Context-sensitive method analysis

```
String appendLn(String s) { return s + "\n"; }
```

```
16 String x = appendLn("Your name is:");  
17 String y = appendLn(sanitizedInput);
```

- use different method types for different calls of the method

call site *type*

16 $appendLn : \text{String}_{[\epsilon]} \rightarrow \text{String}_{[\epsilon]}$

17 $appendLn : \text{String}_{[C_1]} \rightarrow \text{String}_{[C_1]}$

Context-sensitive method analysis

```
15 String doSomething(String sanitizedInput) {  
16     String x = appendLn("Your name is:");  
17     String y = appendLn(sanitizedInput);  
18 }
```

```
54 doSomething(escapeToHtml(input));
```

```
80 doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack *type*

16 :: 54 $appendLn : String_{[\epsilon]} \rightarrow String_{[\epsilon]}$

17 :: 54 $appendLn : String_{[C_1]} \rightarrow String_{[C_1]}$

Context-sensitive method analysis

```
15 String doSomething(String sanitizedInput) {  
16     String x = appendLn("Your name is:");  
17     String y = appendLn(sanitizedInput);  
18 }
```

```
54 doSomething(escapeToHtml(input));
```

```
80 doSomething(escapeToJs(input));
```

- use different method types for different calls of the method

call site stack *type*

16 :: 54 $appendLn : String_{[\epsilon]} \rightarrow String_{[\epsilon]}$

17 :: 54 $appendLn : String_{[C_1]} \rightarrow String_{[C_1]}$

16 :: 80 $appendLn : String_{[\epsilon]} \rightarrow String_{[\epsilon]}$

17 :: 80 $appendLn : String_{[C_2]} \rightarrow String_{[C_2]}$

Context-sensitive method analysis

```
15 String doSomething(String sanitizedInput) {  
16     String x = appendLn("Your name is:");  
17     String y = appendLn(sanitizedInput);  
18 }
```

```
54 doSomething(escapeToHtml(input));
```

```
80 doSomething(escapeToJs(input));
```

- distinguish method types by call context from finite set Cxt
- method call rule relies on context switch function

$$\phi : Cxt \times Cls \times Mtd \times PP \rightarrow Cxt$$

to select the method type for the invoked method

- inference generates new method type for unseen contexts

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

Region types

refine class types with sets of regions:

$$\Gamma \vdash e : C_R$$

regions further classify objects:

- represent disjoint sets of possible concrete memory locations
- e is a location that is in one of the regions in R
- two locations typed with disjoint sets do not alias

increases typing precision:

$$\begin{array}{ll}fty(\text{List}, r, \text{elem}) = \text{String}_{\{\{Input\}\}} & fty(\text{List}, s, \text{elem}) = \text{String}_{\{\{C1\}\}} \\fty(\text{List}, r, \text{next}) = \text{List}_{\{r\}} & fty(\text{List}, s, \text{next}) = \text{List}_{\{s\}}\end{array}$$

Region type system

- type system parametrized with abstraction principles from pointer analysis
 - choice of regions for new objects
 - contexts
- inference: follows existing analysis [Whaley & Lam '04]
 - generate constraints for points-to relations as Datalog rules
 - use external, highly optimized solver
 - interpret resulting relations as types and verify them with type system
- for more information: see LPAR paper